(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
17 May 2001 (17.05.2001)

PCT

(10) International Publication Number
WO 01/35209 A2

(51) International Patent Classification⁷: G06F 9/00

(21) International Application Number: PCT/CA00/01339

(22) International Filing Date: 9 November 2000 (09.11.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/436,995 9 November 1999 (09.11.1999) US

(71) Applicant *(for all designated States except US)*: UNIVERSITY OF VICTORIA INNOVATION AND DEVELOPMENT CORPORATION [CA/CA]; P.O. Box 3075 STN CSC, R-Hut McKenzie Avenue, Victoria, British Columbia V8W 3W2 (CA).

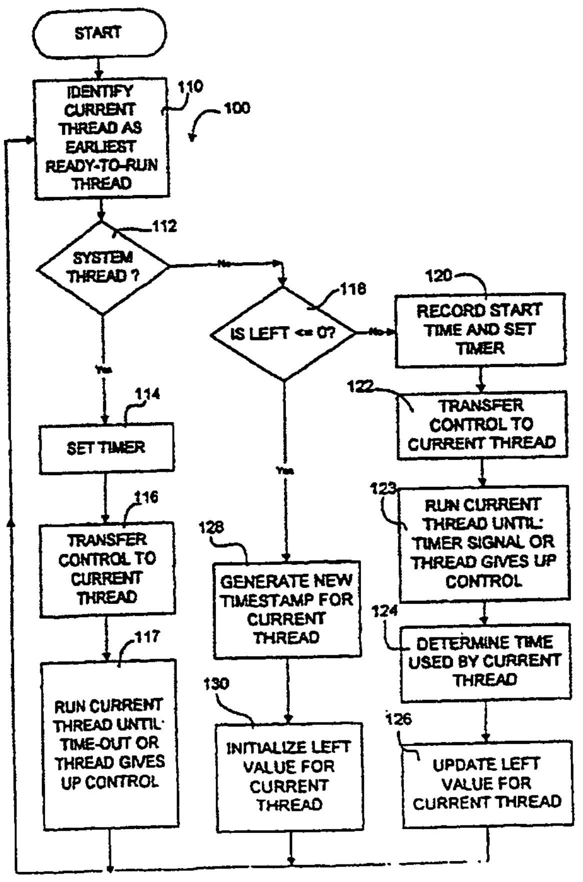
(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: PANG, James, C. [CA/CA]; 1703 - 950 Cambie Street, Vancouver, British Columbia V6B 5X5 (CA). SHOJA, Gholamali, C. [CA/CA]; 1650 Barksdale Drive, Victoria, British Columbia V8N 4Z8 (CA). MANNING, Eric, G. [CA/CA]; 2909 Phyllis Street, Victoria, British Columbia V8N 4Z8 (CA).

(74) Agent: MANNING, Gavin, N.; Oyen Wiggs Green & Mutala, 480 - 601 West Cordova Street, Vancouver, British Columbia V6B 1G1 (CA).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

*[Continued on next page]*

# (54) Title: MODIFIED MOVE TO REAR LIST SYSTEM AND METHODS FOR THREAD SCHEDULING


START
IDENTIFY CURRENT THREAD AS EARLIEST READY-TO-RUN THREAD
110
100
SYSTEM THREAD ?
112
No
Yes
114
SET TIMER
116
TRANSFER CONTROL TO CURRENT THREAD
117
RUN CURRENT THREAD UNTIL TIME-OUT OR THREAD GIVES UP CONTROL
IS LEFT <= 0?
118
Yes
No
128
GENERATE NEW TIMESTAMP FOR CURRENT THREAD
130
INITIALIZE LEFT VALUE FOR CURRENT THREAD
120
RECORD START TIME AND SET TIMER
122
TRANSFER CONTROL TO CURRENT THREAD
123
RUN CURRENT THREAD UNTIL TIMER SIGNAL OR THREAD GIVES UP CONTROL
124
DETERMINE TIME USED BY CURRENT THREAD
126
UPDATE LEFT VALUE FOR CURRENT THREAD

(57) Abstract: Methods and systems for scheduling threads in a multi-threaded computer system use a modified move-to-rear list scheduling algorithm. Threads are ordered in a service list according to a virtual time value. System threads always retain a virtual time value. For system threads, the virtual time value serves as a priority. For user threads, the virtual time value is incremented after the thread has received a share of access to the CPU resource. The invention can provide soft real time capability for application software. At the same time, it satisfies system threads which must be executed with some urgency. The thread scheduler of the invention may be used to advantage in the Java multi-threading framework.

WO 01/35209 A2

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

**Published:**
— *Without international search report and to be republished upon receipt of that report.*

*For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.*

# MODIFIED MOVE TO REAR LIST SYSTEM AND METHODS FOR THREAD SCHEDULING

## Technical Field

This invention relates to scheduling of threads in a multi-threaded computer system. The invention has particular application to managing CPU resources on the JAVA platform.

## Background

Modern general purpose computers make it possible to provide a wide variety of applications which are multimedia in nature. A multimedia application must be run on an underlying supporting environment which can properly support the delivery of continuous media data. Otherwise the multimedia application will not be presented in a satisfactory way. Real-time requirements are imposed on the host operating system and its subsystems because continuous media data, such as digital audio or digital video, must be presented continuously at a predetermined rate in order to correctly represent the information being carried. The requirements on the operating system may be classified as being "soft real-time" because, in general, the operating system is only required to statistically guarantee that quality of service parameters, such as delay and throughput, will be maintained. Multimedia applications often have deadlines for completing various tasks. Fortunately, missing a particular deadline is generally not fatal as long as the deadline is not missed by too much and as long as most other deadlines are not missed.

Multimedia applications are typically very resource intensive. A supporting computer platform must be able to partition, monitor and police the usage of system resources so that all current application programs can make adequate progress, even in the presence of heavy system loads.

One platform that has several desirable characteristics for multimedia computing is the Java platform. In the Java platform, application software written in the Java programming language runs on top of a Java language programming environment. The Java language programming environment includes a Java operating system. The Java operating system runs either on computing hardware which includes a Java processor which directly interprets Java language commands (sometimes known as Java on Java) or on conventional general purpose computing hardware running an operating system, such as Windows NT or UNIX, and a software Java processor emulator (also known as a Java virtual machine) running on the operating system.

Programmers write applications for the Java platform within the Java language programming environment. The Java programming language has a number of advantages. It is small and easily-understood to programmers. Java is object oriented, has built-in support for multi-threaded programming and has automatic memory management. Java is portable, dynamic and robust. In addition an extensive collection of software libraries including Java class libraries and OS specific "native" libraries is available to programmers. The fact that a wide selection of software libraries is available facilitates the rapid development of a wide variety of applications including multimedia processing applications. The interpreted nature of Java enables Java applications to run on a wide variety of computer hardware and a wide variety of operating systems without modification.

A problem with implementing multimedia applications on Java platforms is that Java presently does not provide support for real-time processing. Java does not provide any mechanism which can be used to monitor, manage or police the usage of the CPU resource to ensure the proper delivery of continuous media data. Current implementations of Java use a static priority-based algorithm for thread scheduling. Such algorithms have been shown to be largely ineffective for multimedia applications.

A further problem is that the Java product specification does not specify how thread scheduling should be implemented. Different thread scheduling mechanisms are used by various Java virtual machines. This leads to inconsistencies across platforms and can cause Java applications which function properly on one platform to not function properly on other platforms.

Nilsen Adding Real Time Capabilities to Java, *Communications of the ACM,* 41(6):49-56, 1998 proposes certain extensions to the Java environment which are designed to facilitate real-time computing. The Nilsen system relies on an off-line analyzer and scheduler to provide information to a real-time executive for deterministic execution. The Nilsen system is not fully compatible with the Java reference implementation and requires applications to be modified and specifically optimized. The Nilsen system is therefore not useful for general purposes.

Sun Microsystems has recently proposed a real-time extension to the Java platform. The proposed extension relies on static priority-based scheduling and exploits real-time facilities in underlying real-time operating systems. Unfortunately, hard real-time scheduling based upon static priorities has been shown to be inefficient for running soft real-time multimedia applications.

Various algorithms suitable for scheduling threads in multi-threaded computer systems are known. Many of these algorithms were developed for scheduling packets in packet switched networks and were subsequently adapted for use in scheduling threads in multi threaded computer systems. For example, weighted fair queuing (WFQ) computes start and finish times for each entity being scheduled and schedules the entities in increasing order of their finish times. The need to compute finish times makes WFQ very computationally intensive. Furthermore, WFQ does not provide fairness when the available bandwidth fluctuates over time due to, for example, sporadic interrupt processing.

Fair queuing based on start time (FQS) is similar to WFQ but schedules entities in increasing order of their start times. FQS is also computationally expensive and does not provide fairness when available bandwidth fluctuates.

Self clocked fair queuing (SCFQ) functions in a similar manner to WFQ but uses an approximation to calculate the finish times for each entity. SCFQ is quite efficient. Unfortunately, SCFQ achieves efficiency at the expense of a maximum scheduling delay which may be unacceptable for many applications, especially multimedia applications.

The start time fair queuing (SFQ) algorithm assigns start and finish tags to each entity to be scheduled and then schedules the entities in increasing order of their start tags. A disadvantage of SFQ is that its delay bound increases linearly with the number of threads in the system. Thus it is not ideal for a general purpose environment in which the number of threads may vary.

The experimental Plan 9 operating system incorporates a new scheduling algorithm called Move to Rear List Scheduling (MTR-LS). Move to rear scheduling is described in J. Bruno et al., Move-To-Rear List Scheduling: a new scheduling algorithm for providing QoS guarantees, *Proceedings of the Fifth ACM International Multimedia Conference*, pp. 63-73, November 9-13, 1997. The MTR-LS algorithm assigns a weight, called a service fraction, to each thread. The service fraction specifies the minimum amount of the CPU resource to be allocated to the thread in absolute terms as a fraction of the total available CPU resource. Each thread is also assigned a time stamp. Scheduling of threads is based on their time stamps. The time stamps are adjusted according to the service fractions of their respective threads and the amount of CPU resources consumed by the threads. In addition to the normal QoS guarantees, MTR-LS can provide a "cumulative service guarantee".

A disadvantage of the MTR-LS algorithm is that it cannot effectively be used to schedule system threads which use priorities to specify their urgency. In a priority-based system, the system thread that services timers and clock interrupts can be given a priority higher than that of any other threads in the system. In such a priority-based system the scheduler can pre-empt the current thread and schedule the timer and clock service thread as soon as the timer and clock service thread becomes runnable. With a move to rear list scheduling algorithm it is not possible to express the relative urgency of different threads since each thread is merely guaranteed a share of the CPU resource but threads are otherwise equal in priority.

There is a need for systems and methods for scheduling threads in multi-threaded computer systems which provide fairness and quality of service guarantees, and yet retain the ability to effectively schedule system threads which must be executed with some urgency. There is a particular need for such systems and methods which can be applied to the Java operating system because the Java operating system is otherwise well adapted for use in multimedia applications.

Summary of the Invention

This invention provides systems and methods which use a modified move to rear list scheduling algorithm. In the modified move to rear list scheduling algorithm each thread is assigned a timestamp value. For high priority system threads the timestamp value is set so that it is always earlier than timestamp values for most other threads. In preferred embodiments the timestamp values for system threads are static. The time stamp values for low priority system threads, such as idler threads are set so that they are always later than the time stamp values of most other threads. The value of the timestamp for the high priority system threads indicates a high priority with which the system threads should be executed if they are ready to run. Conversely, the time stamps of low priority system threads indicate a low priority. All user threads have time stamps that are between those of the high priority system threads and the low priority system threads. For user threads, the timestamp value is reassigned each time the thread uses up an amount of the CPU resource which has been allocated to the thread.

One aspect of the invention provides a computer implemented method for scheduling the running of threads in a multi-threaded computer system. The method comprises maintaining a list of a plurality of threads, each having a timestamp value. A scheduler identifies, as the current thread, the thread which has the earliest timestamp value of any threads which are ready to run. The method maintains a time left value for each thread. If the time left value for the current thread indicates that the thread has some time left to run then control is transferred to the current thread for a running time not exceeding the time left value for the current thread. After the current thread has run for the running time, if the current thread is not a system thread, then the running time is subtracted from the time left value for the thread. Whenever the time left value for the current thread is not greater than zero, the current thread is assigned a new timestamp value which is greater than the timestamp values of any of the other user threads.

The time left value for the current thread is then re-initialized as described below.

The methods of the invention permit the fair scheduling of threads while enabling system threads to be scheduled in a manner which takes into account the urgency of the system threads relative to other system threads and user threads.

Another aspect of the invention provides a multi-threaded computer system which implements modified move to rear list thread scheduling. The computer system includes: a processor; and a multi-threaded operating system running on the processor. The operating system comprises a plurality of system threads.. User software on the computer system comprises one or more user threads. The system includes a memory accessible to the processor. The memory contains a data structure comprising a record for each of a plurality of threads. The plurality of threads comprising at least one high priority system thread and the one or more user threads. Each record includes a timestamp value, a service fraction value and a time left value for the thread corresponding to the record. The computer system includes a scheduler which is adapted to identify as a current thread one of the plurality of threads which is ready to run and has a timestamp value earlier than the timestamp value of any other of the plurality of threads which is ready to run. If the time left value for the current thread is greater than zero, the scheduler transfers control of the processor to the current thread for a running time not exceeding the time left value for the current thread; and, if the current thread is not a system thread, the scheduler subtracts the running time from the time left value for the current thread. If the time left value for the current thread is not greater than zero then the scheduler assigns a new timestamp value to the current thread, the new timestamp value being later than the timestamp values of any of the user threads. The scheduler then initializes the time left value of the current thread and selects a new current thread.

Further aspects, advantages and inventive features of the invention are described below.

## Brief Description of Drawings

In drawings which illustrate non-limiting embodiments of the invention:

Figure 1A is a schematic view of a conventional prior art computing environment;

Figure 1B is a schematic view of a prior art Java computing environment incorporating a Java virtual machine running in a conventional computing environment;

Figure 1C is a schematic view of a prior art Java environment including Java software running on a computing hardware which includes a Java processor;

Figure 2 is a schematic view of a computer system according to the invention;

Figure 3 is a service list for use in systems and methods according to the invention;

Figure 4 is a flowchart illustrating a method according to the invention;

Figure 5 is a diagram illustrating the assignment of a new timestamp value to a user thread in the method of the invention; and,

Figure 6 is a block diagram illustrating functional units of a computer system which includes a modified MTR-LS thread scheduler according to the invention.

## Description

Figure 1A illustrates a conventional computing environment in which an operating system runs on computer hardware. The computer hardware incorporates one or more processors and provides other resources which may be used by applications. The resources may include memory, access to peripheral devices, and so on. A programming environment mediates interactions between application software and the operating system and interactions between application software and the computer hardware. The application software runs in the programming environment. A problem which has been recognized with the conventional computing environment is that application software must be written specifically for each programming environment since not all programming environments provide the same resources for use by application programs.

Figure 1B shows a prior art Java computing environment. An operating system runs on suitable computing hardware. The operating system may be, for example, Microsoft Windows™ or Sun Solaris™. A Java processor emulator (known as a Java Virtual Machine or "JVM") runs on the operating system. A Java operating system runs on the Java processor emulator. The Java operating system provides resources to a Java language environment. Java application software runs in the Java language environment. Commands which make up the Java application are received by the Java operating system and interpreted by the Java processor emulator. The Java operating system and processor emulator, in turn, cause the operating system and/or hardware to implement the commands. One advantage of the JAVA language environment is that the JAVA language environment is standardized so that every JAVA language environment provides the same resources to JAVA applications without regard to the computing hardware on which the JAVA language environment is provided.

Figure 1C shows an alternative Java computing environment which includes a JAVA processor. A JAVA processor is a microprocessor that is capable of executing directly Java instructions (called "byte codes"). When such a processor is available, the Java operating system and programming environment can be written in the Java language and executed on the processor directly. This eliminates the need for a general purpose operating system and saves much overhead.

The following description explains an embodiment of the invention which provides thread scheduling in a Java computing environment, such as the environment shown in either of Figures 1B or 1C. The invention has particular application to this environment but the methods and systems of the invention may be used for thread scheduling in multi-threaded computer systems generally.

Figure 2 shows a computer system 20 according to the invention. System 20 has a processor 22, a memory 24 accessible to processor 22, and application software 26, which includes a plurality of threads containing instructions to be executed by processor 22. System 20 also includes a software programmable system timer 28 and JAVA operating system software 30 which includes a scheduler 32 which schedules the running of threads of application software 26.

For each active thread, JAVA operating system software 30 maintains a record in a data structure 34 in memory 24. Data structure 34 maintains information about the thread. The form of data structure 34 is not particularly important. Data structure 34 may include information such as the name of the thread; a pointer to stack memory for the thread; a program counter; a register file; machine context information; and a pointer to a parent process as is known in the art. In addition, data structure 34 includes, for each thread, fields for a timestamp value, a service fraction value and a time left value. JAVA operating system 30 also maintains a service list 40, which contains a list of all active threads. Service list 40 is used by scheduler 32 to schedule threads, as described below. In preferred embodiments of the invention, JAVA operating system 30 maintains a queue 42 which contains a list which includes entries for all active threads which are ready to run.

In a computer system 20 according to the invention various types of threads share processor 22. Processor 22 must service high priority system threads. Such threads must be serviced with a high priority for proper operation of computer system 20. An example of a high priority system thread is the thread which services a system clock interrupt and services timeouts generated by timer 28. A second group of threads is the group of threads which make up running user application software 26. Such threads may be called "user threads". The programmers of application software 26 may wish to control how much time processor 22 spends servicing various ones of the user threads. Threads which require large amounts of the processor resource may be given more time to run on processor 22 than other less computationally intensive threads. Finally, computer system 20 may include a number of low priority system threads which perform functions such as running the finalize() routines of discarded JAVA objects or performing garbage collection routines.

As noted above, data structure 34 includes a timestamp value for each thread. As illustrated in Figure 3, threads in each different group of threads are assigned timestamp values in a different range. Figure 3 is a map of all possible timestamp values. In the preferred embodiment of the invention, the space of possible timestamp values is subdivided into four ranges, a high priority system thread space 44, a user thread space 46, a low priority system thread space 48, and a reserved space 50. The user thread space 46 is typically much larger than any of the other thread spaces.

High priority system threads are assigned timestamp values from high priority system thread space 44. Since any practical computer system will have a limited number of high priority system threads, high priority system thread space may include a relatively small number of timestamp values. In the preferred embodiment of the invention a clock handler thread is assigned an earliest timestamp of all and a time slicer thread is assigned the next earliest timestamp.

User threads are assigned timestamps in the range 46. Low priority system threads may be assigned timestamps in the area 48. The reserved space 50 is preferably left available for various system tasks, as described below.

The timestamp values assigned to threads are not directly connected to the time as measured by any clock but are numbers. Smaller numbers may be associated with "early" timestamps and larger numbers with "later" timestamps or vice versa. Preferably the timestamp is represented as an integer which permits the timestamp value to have a very large range of values. For example, a 64-bit integer may be used to represent the timestamp value. In general the timestamp value is preferably an integer of 45-bits or more.

The active threads in service list 40 can be conceptually arranged in order of their timestamp values, from earliest to latest. The position of each active thread in such an ordered service list 40 is determined by the timestamp values for the thread. Threads having earlier timestamps (it can be appreciated that "earlier" may be represented by a higher or lower values of the timestamp) appear near the head of service list 40 (i.e. toward high priority system area 44). Threads having later timestamps appear farther toward the rear of service list 40. Each thread has a service fraction value which indicates how much of the processor's time should be allocated to processing instructions in that thread.

It is not necessary to keep the entries in service list 40 ordered by timestamp value since, at any given time, a large number of threads will not be ready to run. At any given time there will be a large number of active threads which are not ready to run. For example, threads may be waiting for a resource, such as a printer, a disk drive, or some other peripheral to become available; threads may be waiting for other threads to complete certain operations or to supply certain results; and/or threads may be waiting for another thread to leave a monitor. Scheduler 32 is only required to schedule threads which are ready to run. Preferably scheduler 32 maintains in queue 42 a list of threads which are ready to run wherein threads with earlier timestamps are closer to the head of queue 42 than are threads which have later timestamps. Queue 42 preferably employs a heap data structure for efficiency.

As shown in Figure 4, a method 100 for scheduling threads according to the invention begins by identifying a current thread (Step 110). The current thread is the thread in queue 42 which has a timestamp which is earlier than the timestamps of any other threads in queue 42. Where threads in queue 42 are sorted in order of timestamp as described above then scheduler 32 can then be implemented by simply selecting the first thread in the ready to run queue as the current thread.

System threads are handled differently from user threads. Scheduler 32 determines if the selected thread is a system thread (step 112). If the thread identified as the current thread is a system thread then timer 28 is set to measure an interval which determines how long the system thread will be permitted to run (Step 114). Typically the interval is set to equal either a preemption interval or the value of time left specified in data structure 34 for the system thread, whichever is lower. Scheduler 32 then transfers control to the current system thread (Step 116) and the current system thread is allowed to run (step 117) until timer 28 times out or the current system thread voluntarily relinquishes control over processor 22.

Method 100 then returns to Step 110 to identify a new current thread, which may be the same as or different from the thread which has just executed. The time left value for system threads is not altered.

Each time a running thread is suspended, context switching code, which is included in scheduler 32, temporarily disables any further signals, calculates the CPU resource consumption for the running thread since the context was switched to it and saves the context of the running thread.

If the current thread is not a system thread then a determination is made at Step 118 as to whether or not the time left value for the current thread is greater than zero. If the time left value is greater than zero then the start time of the thread is recorded and timer 28 is set to time out after an interval determined by the shorter of the preemption interval and the time left value for the current thread (Step 120). Once again, timer 28 measures an interval which determines the maximum amount of time that the thread will have to execute without interruption. Each time a user thread begins to run, context switching code records the start time of the thread.

Method 100 then transfers control to the current user thread (Step 122) and permits the user thread to run until either the thread gives up control or timer 28 times out (Step 123). After the thread has completed execution, the time used by the current thread is determined in Step 124. In Step 126 the time left value for the current thread is then updated by subtracting the time used by the current thread (as determined in Step 124) from the time left value for the current thread.

Preferably, if the thread stops running by giving up control before timer 28 times out then at step 126 the time left value for the thread is reduced to zero. As a result, the next time the thread comes to the attention of scheduler 32 it will be moved to the rear of queue 42, as described below, instead of being re-scheduled immediately. A thread may give up control over processor 22 voluntarily when it has finished its current batch of work. A thread may also voluntarily give up control over processor

22 when it is blocked. A thread is blocked when it has further work to do but requires access to some system resource to perform the work. For example, a thread might be blocked when it performs a read from a file and there is no data available, or it performs a monitor entry operation but the monitor is already occupied by another thread.

When a thread is blocked it is no longer "ready to run" and is therefore no longer eligible to participate in the competition among the threads in queue 42 to become the next current thread. Where a thread gives up control of processor 22 because it is finished all of the work that it has to do then it is generally preferable to set the time left value for the thread to zero. If the time left value for the thread is left at a value greater than zero then the thread will still have the earliest time stamp of all user threads, and will be re-scheduled immediately. Resetting the "left" value for a thread which gives up control of processor 22 to zero will ensure that the thread is moved to the rear of queue 42 after all other active user threads in the service list 40. Method 100 then returns to Step 110 to select a new current thread.

If, in Step 118, it is determined that the time left value for the current thread is less than or equal to zero then a new timestamp is generated for the current thread in Step 128. Note that Step 128 is never practised on system threads and therefore the timestamp for system threads remains fixed. In Step 130 the time left value for the current thread is initialized. Initializing the time left value for the current thread typically involves multiplying the service fraction for the current thread by a virtual time quantum *T*. The virtual time quantum *T* is a user-defined system constant. *T* is notionally the time which would be taken for all threads in service list 40 to be executed just once in a case where the service fractions for all of the active threads totals 100%.

As illustrated in Figure 5, as each user thread runs on processor 22, its time left value is reduced. When the time left value for the thread has been reduced to zero then the thread is assigned a new timestamp which is later than time stamps of all other user threads in the service list 40. The result is that threads in the user portion 46 of service list 40 leap frog one another as time passes. In Figure 5, a thread 60 has just completed running. Thread 60 is assigned a new timestamp which is later than the timestamps of any of the four other active user threads in service list 40. Thus, thread 60 is moved to the rear of the list of active service threads, as indicated by arrow 62.

Those threads which have a larger service fraction will have a larger time quantum each time their time left values are initialized in step 130. Consequently, threads having a large service fraction are ahead of other user threads in queue 42 will have more time running on processor 22 before being moved behind other user threads in queue 22 than will user threads having a smaller service fraction.

In the preferred embodiment of the invention, timestamps are not re-used. As each thread uses up the quantum of processor time which was specified when its time left value was initialized the thread is assigned a new timestamp which is later than that of any other active user threads. The integer used to represent the timestamp has a large enough range of values that the supply of new timestamps for user threads is practically inexhaustible. For example, the timestamp may be a 64-bit integer value. In this case, if one hundred values of the timestamp were used up every microsecond then it would still take over 5,000 years to use up every possible timestamp value.

Low priority system threads, which have timestamps in low priority system area 48 will run only when there are no runnable high priority system threads and no runnable user threads since the timestamp values for low priority system threads are later than the timestamp values for any other threads. Low priority system threads might include, for example, an idler thread, a garbage collection thread, and a finalizer thread which invokes the finalize() function of discarded JAVA objects. The idler thread need not do any useful work. The idler thread is always ready to run. Its presence ensures that there is always at least one runnable thread to be scheduled.

Most preferably the garbage collection thread and the finalizer thread are treated as user threads and are each assigned a small service fraction. If these threads are assigned timestamps in low priority system area 48 which gives them very low priorities then they may not run frequently enough if the system is busy running higher priority threads. This can eventually cause a Java virtual machine to run low on memory. If this happens then the Java virtual machine must pause to run the garbage collector thread. If the garbage collector is run as a user thread in a system according to the invention, on the other hand, then the garbage collector can be guaranteed a small share of CPU time without significantly affecting the operation of user threads.

Timestamps in reserved area 50 may be assigned temporarily to user threads for implementing necessary system maintenance tasks which should not be interrupted by user threads. For example, user threads may be temporarily assigned timestamps in reserved area 50 if a low memory situation develops so that it is necessary to run a garbage collection routine. After memory has been freed by running the garbage collection routine then the previous timestamps of the user threads may be restored.

Figure 6 is a block diagram which shows functional units in a computer system according to one embodiment of the invention. System 200 includes a global priority mapper 210, a scheduler 212, a resource consumption tracker 214, a timer handler 216, and a time slicer 218 each of which may comprise a software thread capable of running on a computer processor. Global priority mapper 210 assigns timestamps to user threads and initializes the time left values for each thread according to the service fraction (CPU resource allocation) for the thread.

Scheduler 212 is invoked each time it is necessary to select a new thread to run. The times when scheduler 212 is invoked to select a new current thread may be called decision epochs. A decision epoch occurs, for example, whenever the current thread is blocked or pre-empted. A decision epoch may occur, for example, when a current thread is pre-empted by a signal handler, when the thread performs some system activity, such as I/O, monitor operations or thread creation, or when it yields control of the processor. In a Java environment, scheduler 212 may rely on asynchronous signals from an operating system for notification of events such as timer time out or changes in device status. The handler for such a signal will invoke scheduler 212 to cause the currently running thread to be pre-empted and a new thread (which could be the same thread as the currently running thread) to be scheduled. At each decision epoch, scheduler 212 selects a thread for execution. Scheduler 212 scans list 40 (or queue 42) to locate the runnable thread with the highest global priority (earliest timestamp). If the thread identified is not a system thread and has used up its allotment for the current quantum (i.e. has a time left value $\leq 0$) then scheduler 212 calls global priority mapper 210 to re-initialize the quantum and global priority (timestamp) for the thread. Global priority mapper 310 resets the priority for each thread to be $\alpha \times T + L$ where $\alpha$ is the service fraction of the thread, $T$ is the virtual time quantum and $L$ is the last time left value for the thread. Scheduler 212 then resumes scanning list 40 (or queue 42) to locate another runnable thread with the highest global priority. Scheduler 212 repeats this operation until it finds a thread with a positive time left value which is ready to run.

If the thread identified is a system thread or is a user thread which has not used up all of its allotment for the current quantum then scheduler 212 transfers the control of the CPU to the current thread and sets timer for the time slicer to expire at the end of the next preemption interval, or when the current thread's current quantum is exhausted, which ever comes first.

Resource consumption tracker 214 takes clock readings before the control of the CPU is transferred to a thread and takes clock readings again after the control is given up or revoked. Resource consumption tracker 214 then calculates the CPU resource consumption of the thread during the last execution and, if the thread is not a system thread subtracts this amount from the thread's time left value. Resource consumption tracker may be implemented as part of the context switching code which runs whenever context is switched from one thread to another thread.

Timer handler 216 services hardware clock interrupts and supports timer time-outs. Timer slicer 218 registers timer timeouts with timer handler 216. When the timer times out then time slicer 218 is invoked (or "woken up") and causes re-scheduling of threads. This prevents any thread from monopolizing the use of the CPU resource. Time slicer 218 interrupts the currently running thread after a time ΔT which is equal to the smaller of the time left value for the thread and a preemption interval. Every time scheduler 212 selects a new thread to run it sets a time-out for time slicer 218 that will expire after ΔT time units. When the time-out expires, time slicer 218 becomes runnable. Because time slicer 218 always has an artificially early timestamp value, the time slicer thread pre-empts the current user thread and causes a re-scheduling. If the current user thread is blocked for some reason, such as a monitor operation, before the time ΔT passes then scheduler 212 cancels the time-out and sets out a new time-out when the next thread is scheduled.

When a new user thread is created the new user thread is preferably assigned a timestamp which is later than that of any other user thread in service list 40. Consequently, even though creating the new thread will lead to a decision epoch, it alone will not cause the current thread to be pre-empted.

In this invention Hoare's "monitor" mechanism (as described in C.A.R. Hoare, Monitors: An Operating System Structuring Concept, *Communications of the ACM*, 17(10):549-557, October, 1974) may be used for thread synchronization. When this is done, signal handling is asynchronous and is split into two parts: a very simple signal handling function is invoked when a signal arrives. Ths signal handling function invokes a more elaborate signal handling thread. The primary role of the signal handling function is to notify the signal handling thread when a signal is delivered. When a signal is delivered, the current thread is suspended and the signal handling function for the signal is invoked. The signal handling function first invokes context switching code, as described above, to preserve the state of the current thread. The signal handling function then makes its corresponding signal handling thread ready to run. This may be accomplished, for example, by providing a monitor for each signal handling thread and keeping each signal handling thread in a condition wait queue for its monitor until the signal handling function makes the signal handling thread runnable by setting the condition of the condition wait queue in which the signal handling thread is waiting to true. This removes the signal handling thread from the condition wait queue for its monitor. The signal handling thread becomes runnable immediately. Since the signal handling thread is typically a high priority system thread it will run immediately (unless pre-empted by a higher priority system thread). When the signal handling thread finishes, or is pre-empted, the context switch code is invoked and the scheduler function is called to schedule a new thread. Signal handler functions may use the stack of a pre-empted thread.

As an example, a signal handling function may be associated with a timer 28. The signal handling function is invoked whenever timer 28 times out. The signal handling function can cause a clock handler thread to become runnable, and to perform actions which follow from the time-out.

Preemption by Operating System

One problem can occur when it is desired to implement this invention in a multitasking operating system. If a JAVA virtual machine which implements thread scheduling according to the invention is running in such an operating system then the operating system may periodically preempt the JVM and give the processor to another program. If this happens then the computation of the time used by a thread can become inaccurate. Consider, for example, the situation that would occur if a JVM schedules a thread $T$ at time $t$ and the thread runs until a time $t + \Delta t_1$ at which point the operating system takes away control from the JVM (and thread $T$) until a time $t + \Delta t_1 + \Delta t_2$. Thread $T$ is then allowed to run until time $t + \Delta t_1 + \Delta t_2 + \Delta t_3$. The resource consumption tracker 214 would think that thread $T$ had run for a time $\Delta t_1 + \Delta t_2 + \Delta t_3$ whereas, in reality thread $T$ has only run for $\Delta t_1 + \Delta t_3$ time units.

A solution to this problem is to use the real-time scheduling facility, which is provided by most modern multi-tasking operating systems, to prevent the operating system from arbitrarily preempting the JVM. The JVM may be given a higher priority than any other program running under the multitasking operating system, including some functions of the operating system itself. To ensure that the operating system has access to the processor for necessary tasks, the JVM can run a thread (an "OS thread") which has a service fraction but does nothing but yield control of the processor to the operating system. For example, 10% of the available processor bandwidth might be allocated to the OS thread.

## Timestamp Inversion

Another problem that can occur through the use of the invention is timestamp inversion. Timestamp inversion can occur when monitors are used for synchronization between threads, as is done in Java applications. A monitor is a resource which is available to only one thread at a time. If the monitor is free then a thread can enter the monitor. Otherwise a thread wishing to enter the monitor must wait until the monitor is free. Timestamp inversion can occur when a lower priority thread which is executing inside a monitor blocks a higher priority thread which wishes to enter the same monitor. Consider, for example, the case where $T_1$ and $T_2$ are two threads, $T_1$ is a runnable thread, and $T_1$ has a timestamp earlier that that of any other runnable thread. $T_1$ is running inside a monitor $M$. $T_2$ has a later timestamp than $T_1$ and is waiting to enter $M$. When $T_1$ finishes its quantum i.e. when its "time left" value becomes zero, scheduler 212 will move thread $T_1$ to the rear of queue 42 by assigning it a new timestamp. Since $T_1$ no longer has a timestamp earlier than $T_2$, it should no longer block $T_2$. However, $T_1$ will not give up its resources and in particular it will not give up the monitor $M$ it is currently holding. As a result, $T_2$ cannot proceed, even though it may have an earliest timestamp which is earlier than the timestamp of $T_1$ which occupies the monitor $M$, and is blocking $T_2$ at the entrance of monitor $M$.

The problem of timestamp inversion may be addressed as follows. When a thread attempts to enter an occupied monitor, the thread is blocked. Each time this occurs, the scheduler compares the timestamp for the thread to the timestamp of the thread which "owns" the monitor. If the thread which is waiting to en ter the monitor has an earlier effective timestamp, then the thread which is in the monitor temporarily inherits the earlier timestamp as a new effective timestamp. When the thread in the monitor exits the monitor, the scheduler restores its former timestamp. Moreover, when a thread which is inside a monitor finishes its quantum (i.e. when its time left value becomes zero) that thread is assigned a new timestamp and its time left value is re-initialized. However, if the monitor's wait queue is not empty (i.e. there are other threads waiting to enter the monitor) then the thread in the monitor would temporarily be given its original timestamp as its effective timestamp until it exits the monitor. When the thread exits the monitor the new timestamp comes into effect. As a result, the thread in the monitor may finish its critical section as quickly as possible and release the monitor for use by other threads. The thread will resume its rightful place in the service list after it leaves the monitor. While this scheme for addressing timestamp inversion problems is simple and allows fair scheduling even when synchronization is required, it may have an adverse effect on some quality of service guarantees, such as any fairness guarantees because the blocking thread is allowed to "borrow" time from its next quantum.

Systems which implement modified move to rear list scheduling according to the invention can provide a cumulative service guarantee. The delay that a particular process experiences will not accumulate over the lifetime of the process and the service rate perceived by the process will not be less than the admitted service rate, as specified by the service fraction, by more than a constant amount. Thus, such systems are well adapted for use by soft real-time applications, such as the delivery of multimedia information.

The invention may be embodied in a program product. The program product comprising any medium which carries a set of computer-readable signals corresponding to instructions which, when run on a computer, cause the computer to execute a method of the invention. The program product may be distributed in any of a wide variety of forms. The program product may comprise, for example, physical media such as floppy diskettes, CD ROMs, DVDs, hard disk drives, flash RAM or the like or transmission-type media such as digital or analog communication links.

As will be apparent to those skilled in the art in light of the foregoing disclosure, many alterations and modifications are possible in the practice of this invention without departing from the spirit or scope thereof. Accordingly, the scope of the invention is to be construed in accordance with the substance defined by the following claims.

WHAT IS CLAIMED IS:

1. A computer implemented method for scheduling the running of threads, the method comprising:
    a) maintaining a list of a plurality of threads, the plurality of threads comprising at least one high priority system thread and at least one user thread and, for each of the plurality of threads, maintaining a timestamp value, a service fraction value and a time left value;
    b) identifying as a current thread one of the plurality of threads which is ready to run and has a timestamp value earlier than the timestamp value of any other of the plurality of threads which is ready to run;
    c) if the time left value for the current thread is greater than zero,
        i) transferring control to the current thread for a running time not exceeding the time left value for the current thread; and,
        ii) if the current thread is not a system thread, subtracting the running time from the time left value for the current thread; and,
    d) if the time left value for the current thread is not greater than zero, assigning a new timestamp value to the current thread, the new timestamp value being later than the timestamp values of any of the user threads, and initializing the time left value of the current thread.

2. The method of claim 1 wherein the plurality of threads comprises at least one low priority system thread, the low priority system thread having a timestamp value later than a timestamp value of any of the user threads.

3. The method of claim 1 wherein initializing the time left value of the current thread comprises setting the time left value as the service fraction value of the current thread multiplied by a virtual time quantum.

4. The method of claim 3 wherein the virtual time quantum is a time within which each of the plurality of threads would be executed once if the plurality of threads had service fractions totalling one hundred per-cent.

5. The method of claim 1 wherein the threads are threads in a JAVA programming environment.

6. The method of claim 5 wherein the high priority system threads comprise a clock handler thread, the clock handler thread servicing a clock interrupt.

7. The method of claim 6 wherein the high priority system threads comprise a time slicer thread, the time slicer thread having a later timestamp than the clock handler thread.

8. The method of claim 5 comprising providing an Operating System which provides priority-based real-time scheduling; running a Java Virtual Machine on the Operating System at a priority higher than a priority allocated to the Operating System; and providing an OS thread running on the Java Virtual Machine, the OS thread providing CPU resources to the Operating System.

9. The method of claim 5 wherein the user threads comprise a garbage collection thread and a finalizer thread wherein the garbage collection thread runs a garbage collection routine and the finalizer thread runs finalize() routines of discarded JAVA objects.

10. The method of claim 5 wherein one of the user threads comprises an OS thread, the OS thread providing CPU resources to an underlying operating system.

11. The method of claim 1 comprising maintaining a queue containing records of all ready-to-run threads in the list, the records arranged in the queue in order of the timestamp values for the corresponding threads.

12. The method of claim 1 wherein the timestamp comprises an integer having in excess of 45 bits.

13. The method of claim 7 wherein the time slicer thread uses a timer to discontinue execution of a thread after the lesser of a predetermined preemption time and the time left value for the thread.

14. The method of claim 5, comprising running the threads on a computer comprising a processor capable of directly executing Java byte codes.

15. The method of claim 5, comprising running the threads on a computer comprising a processor running an operating system, a computer program emulating a Java processor and a Java operating system.

16. The method of claim 5 wherein the low priority system threads comprise an idler thread, wherein the idler thread is always ready to run.

17. Computer system comprising a processor, a memory accessible to the processor, an operating system and a computer program emulating a Java processor running on the processor, the computer program providing a scheduler for scheduling the running of Java threads on the processor, the scheduler maintaining a list of a plurality of threads, the plurality of threads comprising at least one high priority system thread and at least one user thread and, for each of the plurality of threads, maintaining a timestamp value, a service fraction value and a time left value; the scheduler adapted to:

a) identify as a current thread one of the plurality of threads which is ready to run and has a timestamp value earlier than the timestamp value of any other of the plurality of threads which is ready to run;

c) if the time left value for the current thread is greater than zero,

   i) transfer control to the current thread for a running time not exceeding the time left value for the current thread; and,

ii) if the current thread is not a system thread, subtract the running time from the time left value for the current thread; and,

d) if the time left value for the current thread is not greater than zero, assign a new timestamp value to the current thread, the new timestamp value being later than the timestamp values of any of the user threads, and initializing the time left value of the current thread.

18. A computer readable medium having computer readable program logic recorded thereon, the program logic, when run on a computer, implementing a method comprising:

a) maintaining a list of a plurality of threads, the plurality of threads comprising at least one high priority system thread and at least one user thread and, for each of the plurality of threads, maintaining a timestamp value, a service fraction value and a time left value;

b) identifying as a current thread one of the plurality of threads which is ready to run and has a timestamp value earlier than the timestamp value of any other of the plurality of threads which is ready to run;

c) if the time left value for the current thread is greater than zero,

i) transferring control to the current thread for a running time not exceeding the time left value for the current thread; and,

ii) if the current thread is not a system thread, subtracting the running time from the time left value for the current thread; and,

d) if the time left value for the current thread is not greater than zero, assigning a new timestamp value to the current thread, the new timestamp value being later than the timestamp values of any of the user threads, and initializing the time left value of the current thread.

19. A computer system comprising:

a) a processor;

b) a multi-threaded operating system running on the processor, the operating system comprising a plurality of system threads;

c) user software running on the computer system, the user software comprising one or more user threads;

d) a memory accessible to the processor, the memory containing a data structure comprising a record for each of a plurality of threads, the plurality of threads comprising at least one high priority system thread and the one or more user threads, each record comprising a timestamp value, a service fraction value and a time left value for a thread corresponding to the record;

e) a scheduler, the scheduler adapted to identify as a current thread one of the plurality of threads which is ready to run and has a timestamp value earlier than the timestamp value of any other of the plurality of threads which is ready to run and, if the time left value for the current thread is greater than zero,

   i) transfer control of the processor to the current thread for a running time not exceeding the time left value for the current thread; and,

   ii) if the current thread is not a system thread, subtracting the running time from the time left value for the current thread; or,

   if the time left value for the current thread is not greater than zero, assigning a new timestamp value to the current thread, the new timestamp value being later than the timestamp values of any of the user threads, and initializing the time left value of the current thread.

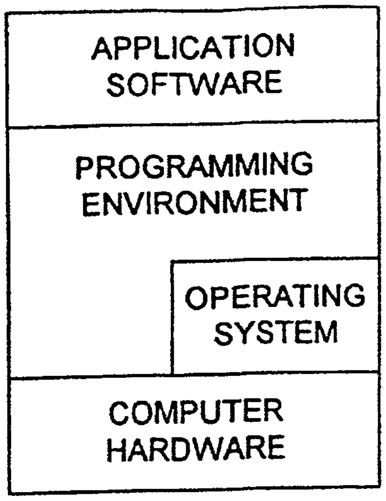


FIG. 1A
(PRIOR ART)

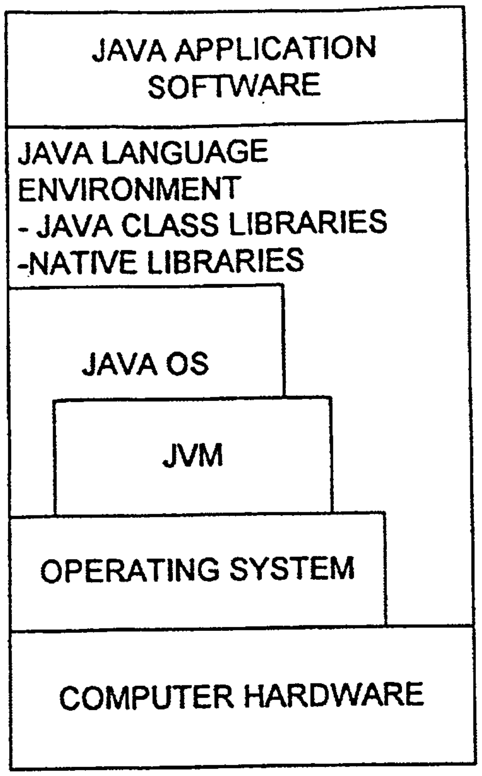


FIG. 1B
(PRIOR ART)

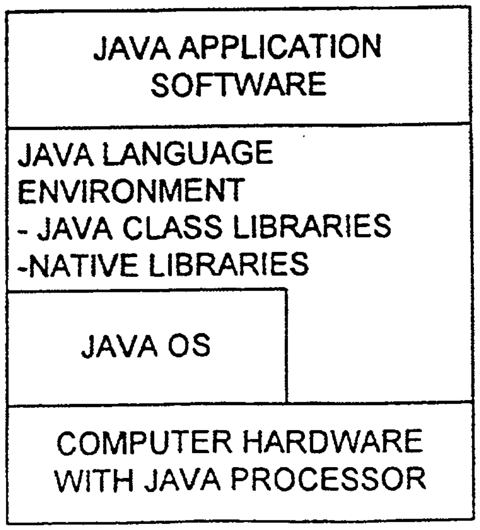

JAVA APPLICATION SOFTWARE
JAVA LANGUAGE ENVIRONMENT
- JAVA CLASS LIBRARIES
-NATIVE LIBRARIES
JAVA OS
COMPUTER HARDWARE WITH JAVA PROCESSOR

FIG. 1C
(PRIOR ART)

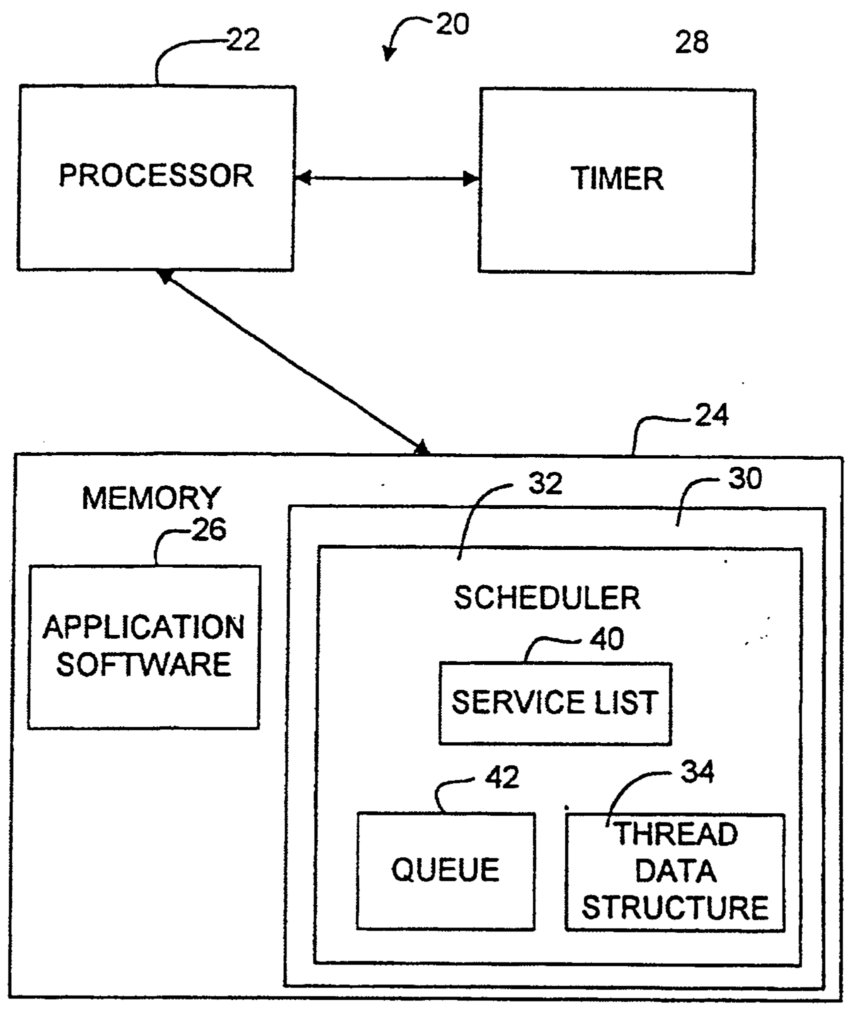

20
22
PROCESSOR
28
TIMER
24
MEMORY
26
APPLICATION SOFTWARE
32
30
SCHEDULER
40
SERVICE LIST
42
QUEUE
34
THREAD DATA STRUCTURE

FIG 2

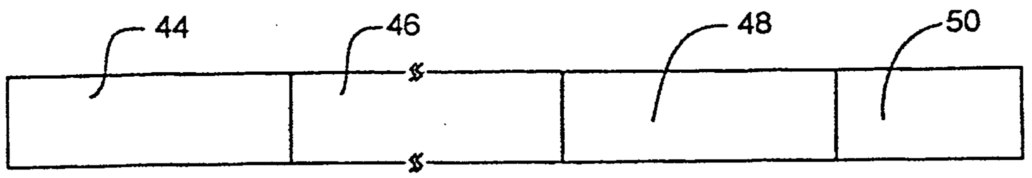


FIG. 3

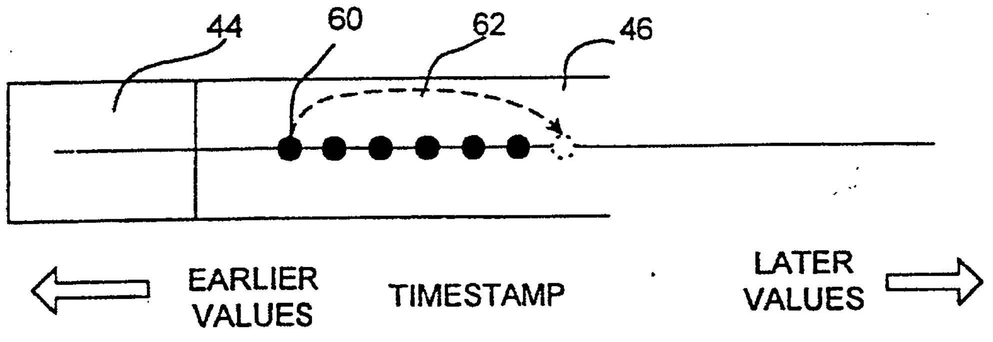


FIG. 5

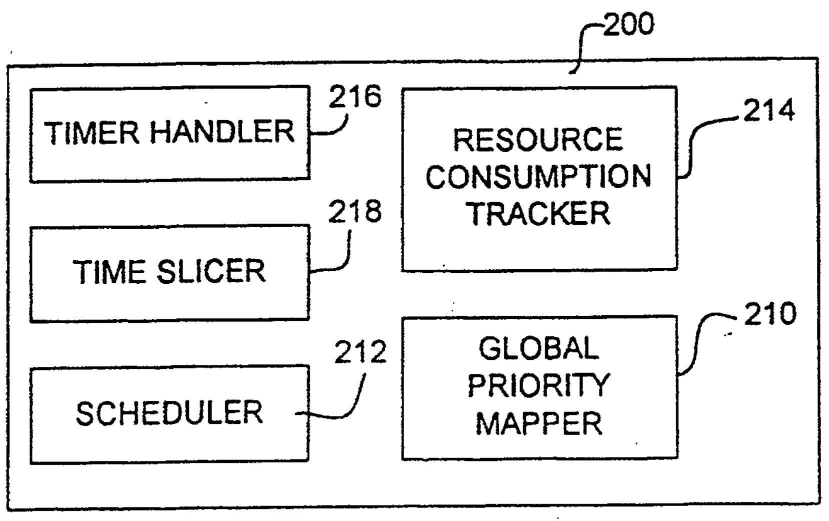


FIG. 6

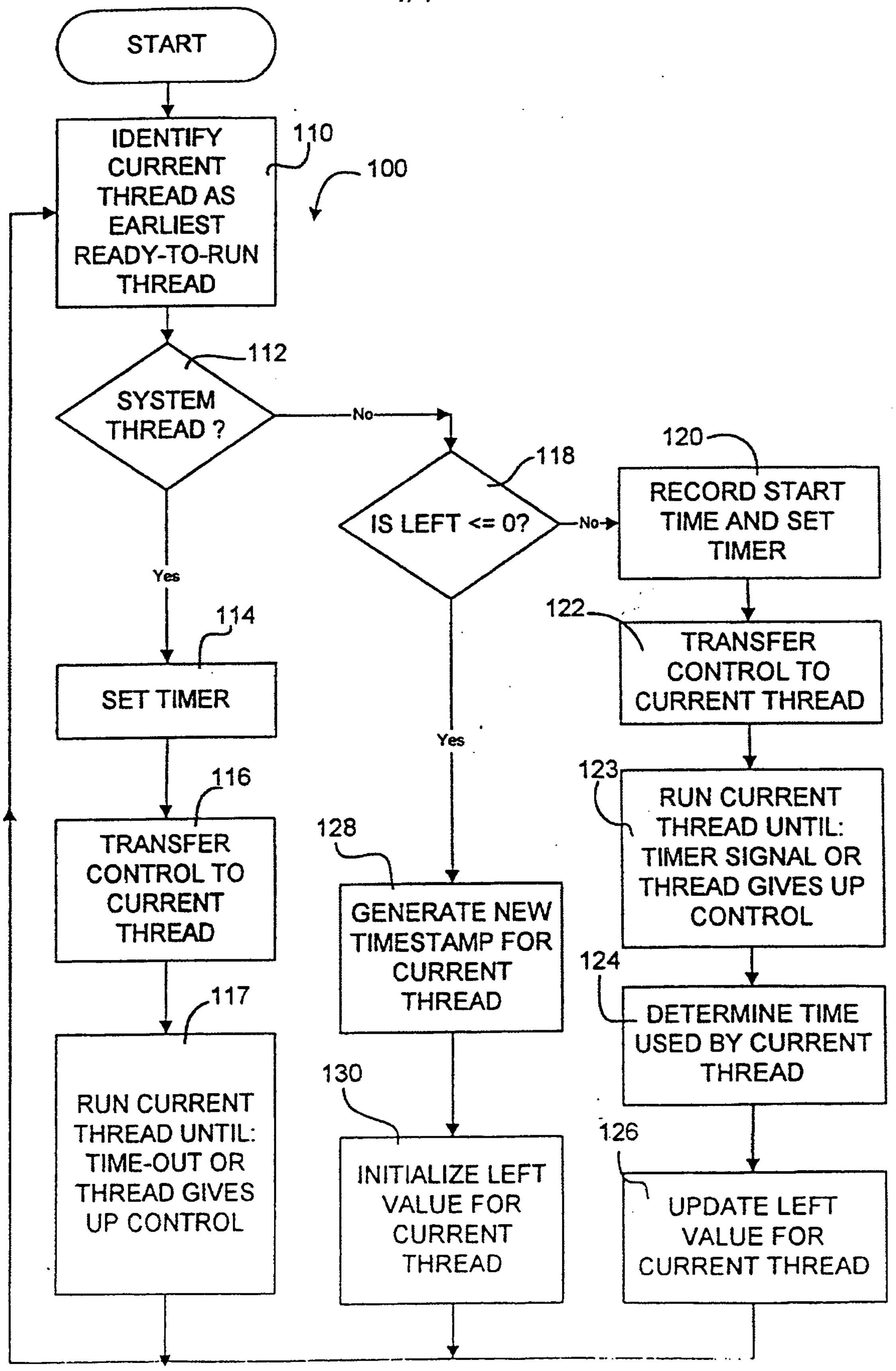

START
IDENTIFY CURRENT THREAD AS EARLIEST READY-TO-RUN THREAD
110
100
SYSTEM THREAD ?
112
No
Yes
IS LEFT <= 0?
118
No
Yes
RECORD START TIME AND SET TIMER
120
SET TIMER
114
TRANSFER CONTROL TO CURRENT THREAD
122
TRANSFER CONTROL TO CURRENT THREAD
116
RUN CURRENT THREAD UNTIL: TIMER SIGNAL OR THREAD GIVES UP CONTROL
123
GENERATE NEW TIMESTAMP FOR CURRENT THREAD
128
RUN CURRENT THREAD UNTIL: TIME-OUT OR THREAD GIVES UP CONTROL
117
DETERMINE TIME USED BY CURRENT THREAD
124
INITIALIZE LEFT VALUE FOR CURRENT THREAD
130
UPDATE LEFT VALUE FOR CURRENT THREAD
126

FIG. 4